J Internet Serv Appl (2012) 3:159–172
DOI 10.1007/s13174-011-0056-5

ORIGINAL PAPER

Towards an opportunistic grid scheduling infrastructure based on tuple spaces

Fábio Favarim · Joni da Silva Fraga · Lau Cheuk Lung

Received: 4 November 2010 / Accepted: 19 December 2011 / Published online: 13 January 2012

Abstract One main issue associated with the efficient and effective use of heterogeneous resources in a grid system is the scheduling. Scheduling in a grid system involves a number of challenging issues mainly due to the dynamic nature of the grid. Schedulers on traditional grid infrastructures rely on an information service that provides information about resources capacities and availability. However, in an asynchronous distributed system like a grid providing up-to-date information about resources is difficult. Current scheduling algorithms make scheduling decisions without fully accurate information about resources which can lead to inefficient schedules. This paper proposes a new scheduling infrastructure for grids where resources select tasks they execute, instead of the traditional approach where schedulers finding resources for the tasks. The new proposed approach allows, at any time, to make scheduling decisions with up-to-date/accurate information. Moreover, our infrastructure provides mechanisms to provide a fault tolerant scheduling. The proposed infrastructure is mainly based on the tuple space coordination model. In our evaluation study, a number of experiments with various simulation setting demonstrated the practicability of proposed infrastructure.

F. Favarim (✉)
Department of Informatics, Tecnological
Federal University of Paraná, Pato Branco, PR, Brazil
e-mail: favarim@utfpr.edu.br

J. da Silva Fraga
Department of Automation and Systems Engineering,
Federal University of Santa Catarina, Florianópolis, SC, Brazil
e-mail: fraga@das.ufsc.br

L.C. Lung
Department of Informatics and Statistics,
Federal University of Santa Catarina, Florianópolis, SC, Brazil
e-mail: lau.lung@inf.ufsc.br

Keywords Grid computing · Scheduling · Fault-tolerance · Tuple space

1 Introduction

Scheduling is an important issue in grid, computing systems. Grid scheduling requires a series of challenging tasks. These include searching for resources in collections of geographically distributed heterogeneous computing systems and making scheduling decisions taking into consideration the quality of service. Despite efforts that current grid schedulers with various scheduling algorithms have made to provide comprehensive and sophisticated functionalities, it has been difficult to guarantee the quality of schedules they produce.

The most challenging issue that they face is the dynamic nature of opportunistic grid environment, that is, the availability and capability of the grid resources change dynamically. Although a resource may be participating in a grid, generally it is not dedicated to it, i.e., its main purpose is for use by local users of the organization that it belongs to, allowing grid applications use the idle time of desktop machines to perform high-performance computation.

Thus, each resource is assumed to be available during a limited time and it is also assumed that typically there are not enough resources to execute all tasks. Schedulers get information about the available resources from an information service [1] and use this information to choose resources for executing the tasks. The decisions a scheduler makes are only as good as the information provided to it. Many theoretical schedulers assume one has 100% of the information needed and that the information is always correct (up-to-date). The process of gathering information about resources is like taking a snapshot of the grid, i.e., getting the

global grid state in a certain instant. This operation is reasonably costly in a large grid, and the snapshot tends to become outdated in a short time when the grid is comprised by a large number of nondedicated, heterogeneous, widely-dispersed resources. Moreover, getting an accurate snapshot in an asynchronous distributed system (as the Internet) has a classical proof of impossibility [2]. The key problem is that information obtained from the information service may be outdated by the time the scheduler needs it to schedule tasks.

Executing computationally intensive applications on dynamic heterogeneous environments such as computational grids, with hundreds, thousands, or even tens of thousands of resources, joins, exits and failures of resources are frequent, can be a difficult task, especially when using nondedicated resources. Unlike dedicated resources, whose mean time between failures is typically weeks, months or even months, nondedicated resources can become unavailable several times during a single day [3]. Moreover, the current grids have single points of failure, i.e., not all their components are fault-tolerant.

In this paper, we propose a novel scheduling infrastructure for grid environments, called GRIDTS. In GRIDTS, the resources select the tasks they want to execute, instead of the traditional infrastructure where schedulers find resources to execute the tasks. Implicitly, this solution does not use an information service and allows scheduling decisions to be done with up-to-date information, since naturally each resource has always up-to-date information about itself. Therefore, our solution overcomes the problems of getting up-to-date information about resources faced by traditional schedulers. Additionally, GRIDTS also provides a fault-tolerant infrastructure, in the sense, all its components can fail by crashing and the system still behaves as expected.

GRIDTS is based on the *generative coordination model*, in which processes interact through a shared memory object called *tuple space* [4]. This coordination model supports communication that is decoupled both in time (processes do not need to be active at the same time) and space (processes do not need to know each others locations or addresses) [5]. This makes it particularly suited for dynamic systems like grids.

This work has two main contributions. Firstly, it presents a new infrastructure for computational grids that allows resources to find tasks suited for their attributes, even if those attributes change with time. Through the correctness proofs and the experimental evaluation, we show that GRIDTS is a highly practicable solution to grid computing. Secondly, the infrastructure provides fault-tolerant scheduling by combining a set of traditional fault tolerance techniques to tolerate crash faults in any component of its infrastructure.

2 Tuple spaces

A *tuple space* can be seen as a shared memory object that allows distributed processes to interact [4]. In this space, generic data structures called *tuples* can be inserted, read, and removed. A tuple t in which all fields have a defined value is called an *entry*. A tuple with one or more undefined fields is called a *template* (usually denoted by a bar, e.g., $\bar{t}$). A tuple space can only store entries, never templates. Templates are used to allow content-addressable access to tuples in the tuple space. An entry t and a template $\bar{t}$ *match* if they have the same number of fields and all defined field values of $\bar{t}$ are equal to the corresponding field values of t. For example, template $\langle$JISA-Journal, 2011, $*\rangle$ matches any tuple with three fields in which JISA-Journal and 2011 are the values of the first and second fields, respectively (the *wild-card* ("*"), represents a undefined field).

A tuple space provides three basic operations [4]: $out(t)$ which inserts a entry t in the tuple space; $in(\bar{t})$ which reads and removes any tuple t which matches $\bar{t}$ in the tuple space; $rd(\bar{t})$ which has a behavior similar to $in(\bar{t})$, but the matched tuple t is not removed from the tuple space. Both *in* and *rd* operations are blocking, i.e., if no tuple matching $\bar{t}$ is available, the process remains blocked until a tuple matching the template $\bar{t}$ being available in the space. Nonblocking versions, *inp* and *rdp*, are also usually provided [4]. These operations work in the same way as their blocking versions but return even if there is no matching tuple for the specified template in the space (signaling failure). In this paper, we also use a variation of the *rd* operation that reads all the tuples that match the template, $copy\_collect(\bar{t})$ [6]. An important characteristic of the generative coordination model is the associative nature of the communication: tuples are not accessed through an address or identifier, but rather by their content.

3 Scheduling in grid environments

Grid scheduling is defined as the process of making scheduling decisions involving resources over multiple administrative domains. This process can include searching multiple administrative domains to use a single machine or scheduling a single job to use multiple resources at a single site or multiple sites. Grid scheduling involves three main phases [7]: resource discovery, system selection, and job execution.

3.1 Resource discovery

The resource discovery phase involves determining the potential set of resources are available to run tasks. The potential of resources is some set that has a minimal feasibility requirement. The set of possible job requirements can be very

broad and will vary significantly between jobs. It normally includes static details (the operating system or hardware for which a binary of the code is available, or the specific architecture for which the code is best suited). Traditional grid schedulers get this statical information from a grid information service. In GRIDTS, this phase is executed by the own resources that gets from tuple space the available tasks to be executed.

3.2 System selection

Given a group of possible resources, all of which meet the minimum requirements for the job, a resource (or resource set) must be selected on which to schedule the job. In order to make the best possible job/resource match, detailed dynamic information (the minimum RAM, processor speed, or /tmp space needed) about the resources is needed. This information is get from the grid information service. With the detailed information gathered, the next step is to decide which resource (or set of resources) to use using some criteria (scheduling heuristics). Various approach are possible, some of them are presented in Sect. 8.1. As in previous phase, in GRIDTS this phase is executed by the resources. Indeed the first two phases are executed in only one step in GRIDTS.

3.3 Job execution

Once resources are chosen, the application can be submitted to the resources. The submission of the application to resources are similar in both infrastructure, by transferring the code and the inputs. When the job is finished, the user needs to be notified. This is normally done sending the results directly to the grid scheduler, and in GRIDTS, it is done by placing results in tuple space.

4 GRIDTS: overview of the infrastructure

Figure 1 presents the GRIDTS infrastructure. GRIDTS uses the tuple space coordination model to support task scheduling. The user submits the application to the broker, which is divided into tasks and then inserts tuples describing tasks to be executed in the tuple space. Grid resources retrieve from the space tuples that describe tasks they are able to execute. After each task execution, the result is placed in the tuple space, becoming available via broker to the user who submitted the tasks to the grid. Each task represents one unit of work that may be performed in parallel with other tasks. The description of a task contains all the required information for its execution such as: the identification of the task, the requirements for its execution (e.g., processor speed, available memory, operating system), the code to be executed,

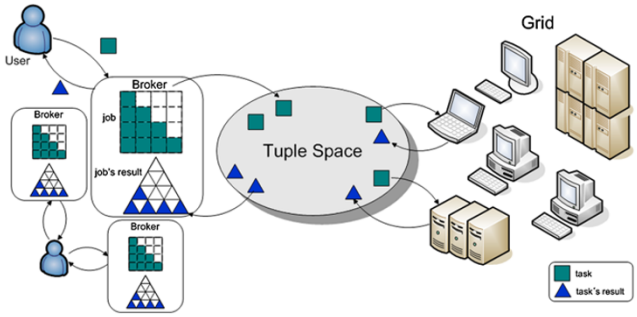

Fig. 1 GRIDTS infrastructure

and the parameters (input data) to the execution of the task. The users do not need to know what resources will execute the tasks, their location, or when these resources will be available.

Scheduling tasks is based on *master-workers* pattern [8]. This pattern has two kinds of entities: one master and several workers. The master send tasks to the workers that execute them and return the results back to the master. However, having all tasks in a grid being managed by only one master could not be a scalable solution. Moreover, if the master fails, all the system could be compromised. In GRIDTS, there is not one but several masters—called *brokers*—that get *jobs* from their *users*, decompose them in *tasks*, and make available these tasks in tuple space to the grid *resources*. When a resource finishes executing a task, it gives the result to the broker, through tuple space, that assembles all results and returns them to the user. All communication between brokers and resources are decoupled, i.e., it is done exclusively through the tuple space. On *master-workers* pattern, the master knows the slaves and the number of them that are available to execute tasks. Differently, in GRIDTS, brokers do not know which and even the number of resources that are available to execute tasks.

The scheduling task model provided by GRIDTS takes from broker the concern to know what resources the tasks will be executed. Moreover, GRIDTS has the immediate benefit of not requiring an information service for indicating the resource utilization. On the contrary, it leverages naturally the scheduling completely decentralized and it enforces a natural form of load balancing since the resources pick tasks adequate to their conditions and get a new one whenever the previous ended.

To provide this new type of scheduling, GRIDTS must deal with two challenges. The first is a fairness problem since multiple brokers can put tasks concurrently in the tuple space. The second is related to the robustness of the scheduling, i.e., making it fault tolerant. Traditional scheduling infrastructures have single point of failures, i.e., some of its components are not fault tolerant. Thus, even brokers, resources, and servers that implement tuple space fail,

GRIDTS has to deal efficiently with these failures, providing the service correctly. The next sections show how these two challenges are treated in GRIDTS.

5 System model

When designing systems, some assumptions are considered about the environment in which these systems are implemented. These assumptions compose the system model. Thus, we describe the assumptions we make throughout the rest of the paper.

5.1 Resource model

We consider a grid $\mathcal{G}$ that consists of a number of sites in each of which a set of computational resources is participating in a grid. Formally, $\mathcal{G} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_g\}$ and $\mathcal{S}_i = \{\mathcal{R}_{i,1}, \mathcal{R}_{i,2}, \ldots, \mathcal{R}_{i,n}\}$ where $\mathcal{S}_i$ is the ith site participating in $\mathcal{G}$, and $\mathcal{R}_i$ is a set of resources at $\mathcal{S}_i$. Let $\mathcal{R}_\mathcal{G}$ denote all resources in $\mathcal{G}$.

Each site is an autonomous administrative domain that has its own local users who use the resources in it. These sites are connected with each other through WAN. Each resource is characterized, at least, by two attributes: speed and load. The *speed* of a resource is the number of instructions computed per unit time. Resources are not entirely dedicated to the grid. In other words, they are used for both local and grid jobs. Each of these resources has one or more processors: memory, disk, etc. The speed of each resource varies over time due the *load* by the original users (resource owner). That is, the speed of each resource is the computing power of the resource which is not used by the original user and is dedicated to the grid.

5.2 Application model

We consider a job $\mathcal{J}$ that consists of a number of n independent tasks τ that have no intertask communications or task dependencies, and thus are suitable for grid implementations. Generally, this type of job is called a bag-of-task (BoT) application [9]. This type of applications exists in many scientific and engineering fields. Formally, $\mathcal{J} = \tau_1, \tau_2, \ldots, \tau_t$ e and $\tau_k = (length, input, code)$, where τ_k is the $k$thtask that composes $\mathcal{J}$. *Length* is the number of instructions (computational cost) of the task τ_k. The length of each task is known and it varies between tasks. *Input* corresponds to the input data needed to execute the task τ_k and *code* represents the code of the task to be executed by the resource. We consider that tasks are computationally intensive, that is, the input data transfer for each task is negligible. The code of the task is also small, and thus transferring it does not influence much to the completion time of the task. Hereafter, the terms application and job are used interchangeably.

5.3 Scheduling model

The grid scheduling problem addressed in this paper is task scheduling of a set $\mathcal{J}$ of n independent tasks, comprising a bag-of-tasks application, onto $|\mathcal{R}_\mathcal{G}|$ heterogeneous resources dispersed across multiples administrative domains in a computational grid. The primary goal of the bag-of-tasks application scheduling is to minimize the makespan. The *makespan* of a job $\mathcal{J}$ is defined as the elapsed time from the time the first task of job $\mathcal{J}$ starts running to the time the last task of job $\mathcal{J}$ completes its execution.

5.4 Interaction model

The client processes (brokers and resources) can only interact through the tuple space. Tuple Space is implemented by DEPSPACE [10], a dependable tuple space. As we assume the *bag-of-tasks* application model—tasks of an application are executed independently—therefore there is no communication between resources and between brokers. We assume that j tuple spaces can exist in the system. Formally, $\mathcal{TS} = \{ts_1, ts_2, \ldots, ts_j\}$, where each tuple space ts is implemented by a set of n *servers* $U = \{s_1, s_2, \ldots, s_n\}$. Each tuple space ts is defined to be accessed by any set of client process. The client process of $\mathcal{TS}$ is divided in two subsets: the resources set $|\mathcal{R}_\mathcal{G}|$ and the brokers set $\mathcal{B} = \{b_1, b_2, \ldots, b_l\}$.

GRIDTS does not require any explicit time assumption, i.e., there are no time bounds on the communication and processing delays. However, since we use the DEPSPACE and it is built on a total order multicast primitive based on the Byzantine Paxos consensus protocol (PBFT) [11], ensuring that all replicas execute the same sequence of operations, an *eventually synchronous system* behavior [12] is required for liveness. Moreover, the eventually synchronous model also is important for transaction support mechanisms, which only can ensure the ACID properties (Atomicity, Consistency, Isolation, Durability) [13] in times when the system has a synchronous behavior.

5.5 Fault model

We assume that an arbitrary number of clients of $\mathcal{TS}$ (brokers and resources) can be subject to *crash failures*. Each process behaves according to its specification until it possibly crashes or stops executing for some reason. A process that never crashes is said to be *correct*, while a process that crashes is said to be *faulty*. It is important to clarify that the abnormal termination of a resource is not the only reason for a task to stop being executed. If a resource becomes unavailable to the grid for any reason, for instance, because its owner needs to use it, then we also consider it as a crash of the resource.

Tuple Space (DepSpace) is implemented in replicated in a set of n servers. We assume a bound of up to f these

servers can be subject to *Byzantine failures*, i.e., they can deviate arbitrarily from the algorithm they are specified to execute and work in collusion to corrupt the system behavior. The fault tolerance to the tuple space is guaranteed using replication [14, 15] over PBFT [11]. It is required $n \geq 3f + 1$ servers to tolerate the aforementioned f faulty servers. We assume *independence of faults* for servers, which can be substantiated in practice using several types of diversity [16].

6 GridTS properties

Distributed systems have been specified in terms of safety and liveness properties. Informally, a safety property states that "something bad will *not* happen" during the system execution, while a liveness property states that eventually "something good *must* eventually happen."

Consider that a *task ready to be executed* corresponds to a tuple describing the task on the tuple space. There are two properties that have to be satisfied by GridTS:

- *Partial correctness*: if a resource executing a task fails, then the task becomes again ready to be executed.
- *Starvation freedom*: if there is some task ready to be executed and a correct resource able to execute it, then this task will eventually be executed.

The first (safety property) says that a task does not disappear if the resource that is executing it fails. The second (liveness property) says that every task will be executed if there is at least one correct resource capable of executing it, i.e., no task will be waiting forever to be executed. These are the main properties the system has to satisfy to guarantee that all tasks are executed if there is at least one resource that does not fail.

7 Designing GridTS

We begin this section by presenting how the two challenges—fair and fault tolerant scheduling—presented in Sect. 4 are treated. Afterward, we describe the algorithms executed by the brokers and the resources, and its correctness proofs.

7.1 Fair scheduling

In order to guarantee a *fair scheduling*, we propose the *FIFO-Except* scheduling criteria. FIFO-Except is based on the FIFO (First-In-First-Out) scheduling criteria, i.e., first jobs that come to grid will have higher priority, having its tasks executed first of such jobs that have lower priority. However, sometimes FIFO criteria cannot be placed, because there may be tasks that require resources other than those currently available in the grid. That is, the resource needs to have the minimum requirements specified by the task in order to be able to execute it. For example, the resource needs to have the operational system, an amount of free memory, and disk space specified, among other requirements. Thus, in these cases, such tasks need to wait until resources that meet the its requirements become available. Thus, only in this case, tasks of jobs with lower priority will be executed before those with higher priority. This is why we have named the criteria as FIFO-Except: it behaves like FIFO, *except* when there are no resources to meet the requirements of the tasks with higher priority. This criteria also allows GridTS to satisfy the starvation-freedom property, which is proven in Sect. 7.5.

Other scheduling criteria, in addition to the FIFO-Except, can be used in GridTS. For example, we can employ a criteria based in a network of favors [17], where users who donate more resources will have greater priority when they need to make use of the grid. This stimulates the users to donate their idle resources to execute applications of others users and to minimize the free-riding users—users that consume resources donated by others without any contributions of their own.

The FIFO-Except scheduling criteria is supported in GridTS by setting a sequencer (*ticket*). The *ticket* in GridTS is implemented through a tuple (ticket tuple) in the tuple space. This tuple contains the value that has not yet allocated to the sequencer *ticket*. In GridTS, in addition to the tasks and the sequencer being represented by tuples in the tuple space, the jobs are also represented by tuples in the space (job tuple). The definition of all tuples used in GridTS is presented in Sect. 7.4. When a broker wants to insert a tuple in the space, it needs to: (i) remove the tuple that represents the sequencer (*ticket tuple*) from space; (ii) inserts in the space the job tuple with the current ticket value; (iii) increases the ticket value and inserts a new ticket tuple, with the ticket value incremented, back in the space. In order to guarantee the FIFO-Except scheduling criteria, resources must pick the task of the job with the lowest ticket value. It can happen that the resource does not meet the requirements of any task of the application with the lowest ticket value, thus it must search for task of applications with highest ticket value.

7.2 Scheduling algorithm

After job selection, the resource needs to select the task of the job to be executed according to some heuristic. It is important to clarify that the performance of scheduling depends strongly on the efficiency of the heuristic chosen. Thus, any scheduling heuristic that uses any or only local information about resources can be used in GridTS. The fundamental ideas of some existing grid scheduling algorithms

could be integrated into the GRIDTS, such as Shortest Job First (SJF), Workqueue (Sect. 8.1). But when the job's tasks are heterogeneous, slow resources might get large tasks, taking much more time to execute them than a faster resource. This could delay the overall job's execution. But we believe that if we proceed in this manner we could lose the great advantage of our proposal that is the knowledge of the availability of the resources. In Sect. 8.1 we present a very simple scheduling algorithm we developed.

7.3 Fault-tolerance

The second challenge faced by GRIDTS—providing a fault-tolerant scheduling infrastructure—is treated by combining three fault tolerance techniques to tolerate crash faults in components of the its infrasctructure: replication, checkpointing and transactions.

7.3.1 Replication

Replication is used to guarantee that the service provided by the tuple space stays available if there are fails, by replicating the server in several computers. We consider that the tuple space is indeed implemented by a set of servers and is fault-tolerant but we do not delve into the details of how it is done since we use DEPSPACE [10] and the replication in tuple spaces area is well understood and essentially solved [10, 15, 18].

7.3.2 Checkpoint

Tasks usually take a long time to execute, e.g., hours or even days, so it is not convenient to restart from scratch the execution of a task whenever the resource that is executing it fails or leave the grid. To minimize this problem, GRIDTS uses a mechanism of backward error recovery that consists in periodically saving the state of the task execution—a checkpoint—in the tuple space [19]. In case of resource failure, another resource can continue the execution of the task from an intermediate execution state contained in a previously saved checkpoint, thus limiting the work lost due to resource failures during the execution of a task. The checkpoint saved must be portable, i.e., the state stored in a checkpoint should be recoverable in a machine with a different architecture. The checkpoint portability can be achieved using the application-level checkpointing, which consists in instrumenting an application source code to save its state periodically. Since in application-level checkpointing we manipulate application source code, semantic information regarding the type of data being saved is available both when saving and recovering application data. This semantic information allows the data saved by a process on an architecture to be recovered by a process executing on another architecture. This is achieved in practice, using the serialization mechanism provided by the Java Virtual Machine [20].

7.3.3 Transactions

A transaction is an abstraction that guarantees essentially the atomic execution of a set of operations on a system (in this case, the tuple space). Thus, a transaction takes the tuple space from one consistent state to another, also consistent after executing a sequence of tuple space operations. To ensure the consistency of tuple space, a transaction must ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties [13]. Therefore, if a process tries to execute a set of operations in the tuple space, either all operations are reflected correctly in the system (the transaction is committed) or it has no effect at all (the transaction is aborted in one of two ways: the process executing the tuple space operations fails or the client aborts it). Thus, in case of abortion, if some operations have already been executed then the tuple space removes all their effects to guarantee the atomicity [13]. In practice, the detection of a failure works in the following way [21]: when a process starts a transaction, it defines a *lease* to do that transaction, i.e., a time interval during which it will execute the transaction's operations. If the process does not commit during that time, the tuple space assumes that the process failed and aborts the transaction. If the process needs more time to execute the transaction, it needs to renew the lease.

Transactions are used by both brokers and resources. A broker uses transactions to ensure that: (1) the corresponding job's tasks are insert atomically in the space, i.e., either they are all inserted or none is (in case the broker fails during the insertion); (2) the ticket is not lost if the broker removes it and crashes before inserting it back incremented in the space; (3) to get the results of the tasks atomically from the space. These transactions allow also the broker not to be locked waiting until all the tasks are executed, i.e., the broker can leave the system after having placed the tasks into the space and later run again to get the results.

On the resources-side, transactions are used mainly to guarantee when a resource fails during the execution of a task, the task tuple is returned to the space to be eventually executed by another resource.

In [22], we propose a transaction model to be integrated in dependable tuple spaces, such as the DEPSPACE [10]. The proposed model guarantees the ACID properties of transactions in more severe environments, such as those subject to malicious faults (Byzantine faults). The transaction support is provided adding some transaction management operations (Fig. 2) to the original operations of DEPSPACE.

Now we describe briefly how we integrated the transaction mechanism to DEPSPACE. The assurance that the transaction does not violate the isolation property is provided through concurrency control mechanisms in tuple spaces, among the concurrency control mechanisms that have been discussed in the literature. We assumed the pessimistic concurrency model through the use of exclusive locks [23] to

beginTransaction(*timeout*) → (*tid*, *grantedTime*)
Creates a new transaction with expiration time *timeout* and returns an unique transaction identifier *tid* and the expiration time *grantedTime* granted to the transaction. The identifier *tid* is used as an additional parameter on tuple space operations to identify them as belonging to the transaction.

closeTransaction(*tid*) → (*commit or abort*)
Ends a transaction. A *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(*tid*) → (*true or false*)
Aborts the transaction. A *true* return value indicates that the transaction has been successfully canceled; a *false* return value indicates that it already been confirmed or cancelled.

renewTransactionTimeout(*tid*, *timeout*) → (*grantedTime*)
Renews the expiration time. A *grantedTime* return value indicates the granted time to the renewal, with $0 \leq grantedTime \leq timeout$. A zero value indicates that it is not possible to renew the expiration time.

Fig. 2 Transaction management operations in tuple spaces

be provided in DEPSPACE. This model is also used in other related works [18, 21].

Before discussing the semantics of the tuple space operations within transactions, we need to formally define some terms used in this paper. A *transaction* T is a sequence of operations $\langle o_1, o_2, \ldots, o_k \rangle$ such that either all or none of the operations of T take permanent effect (*Atomicity*, *Consistency*, and *Durability*), and no effect of the transaction is perceived by other transactions before it is committed (*Isolation*). We say a transaction T holds a *read-lock* on a tuple t when it is reading the tuple t. When a transaction T is removing a tuple t, it is said the transaction T holds a *remove-lock* on tuple t. We say that two transactions T_1 and T_2 are under *conflict* when one of them tries to get a read or remove-lock on tuple t and the have a remove-lock on tuple t. We say two or more transactions *share* a read-lock when both are holding read-lock on the same tuple t. Given these definitions, we redefine the semantics of the tuple space operations, when executed within a transaction, in the following way:

- *out*(t): an entry t written in the tuple space within a transaction T is visible to other processes only after T successfully commits. However, soon after the execution of the *out* operation, t is visible to the process executing T. If t is removed within T, the entry will not be added in the space and it will not be visible outside the transaction when it commits. Entries inserted within a transaction that aborts are discarded.
- *rd*($\bar{t}$) and *rdp*($\bar{t}$): a read may match either an entry t written under the current transaction T_1 or in the tuple space. Such an entry t read from tuple space may be read in any other concurrent transaction T_2, but cannot be removed by it. If there is no tuple t that matches the template $\bar{t}$ is available, *rd* and *rdp* operations behave in different ways:
  - A *rd*($\bar{t}$) operation waits until an entry matching the template $\bar{t}$ is available in the space;
  - A *rdp*($\bar{t}$) operation only waits for an entry t that matches $\bar{t}$ if there is a conflict with other transaction T_2, i.e., other transaction T_2 is removing an entry t that matches $\bar{t}$. Behaving in this manner, if T_2 aborts, the operation *rdp*($\bar{t}$) of T_1 must be able to read t. Notice that an operation *rdp*($\bar{t}$) can block until other transactions under conflict with it terminates; this happens to ensure the isolation property in a conservative way.
- *in*($\bar{t}$) and *inp*($\bar{t}$): these operations may match either an entry t written under the current transaction T_1 or in the tuple space. Such an entry t removed from tuple space may not be read or removed by any other concurrent transaction T_2. Similarly to *rd* and *rdp*, *in* and *inp* operations only differ from each other in the way they behave when there is no tuple matching $\bar{t}$ available on the space:
  - An *in*($\bar{t}$) operation waits until an entry t matching the template $\bar{t}$ is available in the space;
  - An *inp*($\bar{t}$) operation will only waits for an entry t matching the template $\bar{t}$ if there is a conflict with other transaction T_2, i.e., another transaction T_2 is reading or removing the entry t.

Notice that this semantics allow even nonblocking operations such as *inp*($\bar{t}$) and *rdp*($\bar{t}$) to block waiting for conflicting transactions to commit/abort. This happens due to our *pessimistic concurrency model*, in which an operation only completes inside a transaction if it is ensured that it would be committed in the future (contrary to the *optimistic concurrency model* [24]), and to maintain the Isolation property.

7.4 Algorithmic base of GRIDTS

In this section we define the behavior of the brokers and resources by presenting the algorithms they execute. Table 1 presents the structure of the tuples used in algorithms. In all tuples, the first field is the name of the tuple. Most tuples contains the fields *jobId* and *taskId*, which identify the job and the task that they are associated, respectively.

Job and task tuples described in Table 1 were designed for applications whose tasks execute the same code and each task has different input data. Trivial modifications are needed, for instance, applications whose tasks execute different code, but have the same input data, or applications in which both the code and the input data is different for all tasks.

7.4.1 Broker algorithm

The algorithm executed by the brokers is presented in Algorithm 1. It uses transactions to tolerate broker's faults. The first transaction is used to ensure that the job's tasks are inserted atomically in the space, i.e., either they are all inserted

Table 1 The structure of the tuples in GRIDTS

⟨"*TICKET*", *ticket*⟩—represents the tickets used to enforce an order on task execution. The objective is to guarantee the fairness of the scheduling, i.e., starvation freedom. The *ticket* field contains the ticket number. The tuple space is initialized with a tuple ⟨"*TICKET*", *0*⟩.

⟨"*JOB*", *jobId*, *numberTasks*, *ticket*, *information*, *code*⟩—represents all common information of tasks from the same job. The *nTasks* field contains the number of tasks that compose the job, *ticket* is the ticket value associated with the job, and *information* indicates the attributes for job execution (e.g., the required processor speed, memory, operating system). The field *code* can contain either the code to be executed or a reference to its location (e.g., an URL).

⟨"*TASK*", *jobId*, *taskId*, *information*, *parameters*⟩—represents the task to be executed. The *information* field has attributes for task execution. The *parameters* field contains input data for the task or a reference to its location. A task is uniquely identified by *jobId* and *taskId*.

⟨"*RESULT*", *jobId*, *taskId*, *result*⟩—represents the result of a task execution. The *result* field contains the result or a reference to its location.

⟨"*CHECKPOINT*", *jobId*, *taskId*, *checkpoint*⟩—represents the state of a task after a partial execution, i.e., a checkpoint. If a resource fails during the execution of a task, this checkpoint is used by another resource to continue executing the task. The *checkpoint* field contains the partial state of task computation or a reference.

⟨"*TRANS*", *transId*, *ticket*, *jobId*⟩—indicates the last transaction executed by a broker, since brokers execute a sequence of two transactions. The objective is to prevent the broker from re-executing the same transaction if it fails and recovers. *transId* identifies the last process transaction successfully committed. *ticket* and *jobId* have the usual meanings and are used to specify what the broker was doing when it failed.

Algorithm 1 Broker b_i

```
procedure broker(jobId, information, parameters, code)
 1: transId = 1;
 2: rdp(⟨"TRANS", ?transId, ?ticket, jobId⟩)
 3: if (transId = 1) then
 4:    tasks ← generateTasks(parameters);
 5:    begin transaction
 6:       in(⟨"TICKET", ?ticket⟩);
 7:       out(⟨"TICKET", ++ticket⟩);
 8:       out(⟨"JOB", jobId, nTasks, ticket, information, code⟩);
 9:       for i ← 1 to nTasks do
10:          out(⟨"TASK", jobId, tasks[i].id, task[i].information⟩,
                                                  tasks[i].parameters);
11:       end for
12:       transId = 2;
13:       out(⟨"TRANS", transId, ticket, jobId⟩);
14:    commit transaction
15: end if
16: if (transId = 2) then
17:    begin transaction
18:       for taskId ← 1 to nTasks do
19:          inp(⟨"RESULT", b_i, jobId, taskId, ?r⟩);
20:          result ← result ∪ {r};
21:       end for
22:       in(⟨"TRANS", 2, ticket, jobId⟩)
23:       in(⟨"JOB", jobId, nTasks, ticket, information, code⟩);
24:       deliverToUser(result);
25:    commit transaction
26: end if
```

or none (in case the broker fails during the insertion). The second transaction is used to get results of the job's tasks atomically from the space. These transactions allow also the broker not to be locked waiting until all tasks be executed, i.e., the broker can leave the system after having placed the tasks into the space and later run again to get the results.

The algorithm starts by verifying if the broker has been restarted due to a failure. This is done using the *rdp*() operation (line 2). If this operation does not return a tuple, then $transId = 1$ (line 1), the job is divided in a set of tasks (line 4) and the first transaction is executed (lines 5–14), possibly followed by the execution of the second transaction. Otherwise, only the second transaction is executed (lines 17–25). This second situation happens when the first transaction has been completely executed before (so the *out* in line 13 was inserted in the tuple space) but the second transaction was interrupted due to a failure of the broker (so the *in* operation in line 22 was not executed). Therefore, the objective is to avoid the tasks from being reinserted in the tuple space due to a failure.

First transaction starts by getting, incrementing and writing the *ticket* tuple in the space (lines 6–7). These operations must be done inside a transaction because if the *ticket* tuple is removed from the space and not reinserted, no more jobs can be inserted in the space. After handling the ticket, transaction 1 puts one *job* tuple describing the job in the space, and the corresponding *task* tuples (lines 8–11). Transaction 1 finishes with the insertion of the *trans* tuple in the space, indicating that this transaction was successfully executed (lines 12–13). Transaction 2 gets the results of the tasks from the tuple space (lines 18–21). The *trans* tuple and the *job* tuple are removed from the space (lines 22–23). The result is delivered to the user in a reliable way (line 24).

7.4.2 Resource algorithm

Algorithm 2 describes the behavior of a resource r_i. The algorithm starts by searching all *job* tuples in the space (*copy_collect* operation—line 2) and choosing the most adequate to be executed according to some criteria—*chooseJob*() function (line 3). The criteria used is the FIFO-Except, presented in Sect. 7.1. In this case, the *chooseJob*() returns the job with smallest *ticket* value. Thus, the fair scheduling is guaranteed.

After job selection, the resource selects the task it can execute according to ReTaClasses algorithm (*choose Task* operation—line 5). After a task is chosen a transaction begins (lines 7–11). The task chosen is removed from the tuple space (line 8), executed (*executeTask* operation—line 9) and the result is inserted in the space (line 10). The transaction guarantees that these three operations are done atomically. If the resource fails during the transaction, the *task* tuple

Algorithm 2 Resource r_i

```
procedure resource()
 1: loop
 2:   jobList ← copy_collect("JOB", ∗, ∗, ∗, ∗, ∗, ∗);
 3:   job ← chooseJob(jobList);
 4:   if (job ≠ ⊥) then
 5:     taskId ← chooseTask(job);
 6:     if (taskId ≠ ⊥) then
 7:       begin transaction {gets and executes a task}
 8:         inp("TASK", job.jobId, taskId, ∗, ?parameters);
 9:         result ← executeTask(job.jobId, taskId,
                                 parameters, job.code);
10:         out("RESULT", job.jobId, taskId, result);
11:       commit transaction
12:     end if
13:   end if
14: end loop
function executeTask(jobId, taskId, parameters, code)
15: repeat
16:   begin transaction {partially executes a task}
17:     inp("CHECKPOINT", jobId, taskId, ?checkpoint)
18:     partialResult, checkpoint, taskFinished ←
                 partialExecute(code, parameters, checkpoint);
19:     if not (taskFinished) then
20:       out("CHECKPOINT", jobId, taskId, checkpoint)
21:     end if
22:   commit transaction
23: until (taskFinished)
24: return partialResult
```

is returned to the space and will be eventually executed by another resource. The execution of a task is described in Algorithm 2, lines 15–24. If the resource fails, then another resource or even the same in case it recovers, continues the execution of the task from that checkpoint. GRIDTS uses the tuple space as stable storage, so when a resource is executing a task, it periodically inserts a *checkpoint* tuple in the space. Before the resource starts executing a task, it searches for a *checkpoint* tuple in the space (line 17). If it exists the resource starts executing the task from this checkpoint onward.

It is important to notice the task execution uses a *nested transaction* in lines 16–22. If the resource fails when executing this transaction, the two transactions in the algorithm are aborted. However, the *checkpoint* tuple inserted in the last committed nested transaction (line 20) remains in the tuple space, instead of being removed due to the abortion of the parent transaction.

7.5 Correctness proofs

This section makes an argument that GRIDTS satisfies the two properties in Sect. 6. We start by proving the following lemma.

Lemma 1 *If there is some task ready to be executed and a correct resource able to execute it, then some task will eventually be executed.*

Proof The lemma states that there is "some task ready to be executed,"—which means there are at least two tuples in the space (T is the *taskId* and J is its *jobId*):

$$\mathcal{T}_J = \langle \text{"}JOB\text{"}, J, nTasks_J, ticket_J, information_J, code_J \rangle$$

$$\mathcal{T}_T = \langle \text{"}TASK\text{"}, J, T, information_T, parameters_T \rangle$$

The lemma also states that there is a resource r that can execute T and is correct, i.e., executed Algorithm 2 forever without stopping.

The proof is by contradiction. Assume r does not execute any task after some arbitrary instant t. This is only possible in two situations:

- r blocks at one of the lines 1 to 24. However, an inspection of the algorithm shows that the only line that might block is line 2 since *copy_collect*() is a blocking operation, but this cannot happen since there is at least one job tuple in the space, $\mathcal{T}_J$.
- r does not manage to get a task tuple from the space, which is not possible because there is at least one task tuple in the space, $\mathcal{T}_T$.

This is a contradiction, so some tasks will eventually be executed. □

The following theorems state that GRIDTS satisfies the two properties in Sect. 6:

Theorem 1 *If there is some task ready to be executed and a correct resource able to execute it, then this task will eventually be executed* (*Starvation freedom*).

Proof This theorem is similar to the lemma above but the lemma states that any task is executed, while the theorem is about *this* task.

Let us consider, like in the previous lemma, that the job is described by the job tuple $\mathcal{T}_J$ and the task by the task tuple $\mathcal{T}_T$. The lemma proves that some task is executed. Obviously, we can apply the lemma iteratively to show that infinite tasks are executed. What we have to prove is that task T is not left behind indefinitely. This is enforced by the *ticket* mechanism.

The job of the task to be executed is selected in lines 2–3 by function *chooseJob*() (Algorithm 2). In the text in Sect. 7.1, we stated that this function chooses the job with the smallest ticket value, say, $ticket_{J'}$. We are interested in the case that T has not yet been executed, therefore, $ticket_{J'} \leq ticket_J$. There are two cases:

- if $ticket_{J'} = ticket_J$, then eventually the resource(s) will eventually execute all tasks of J, including T (given the lemma).
- if $ticket_{J'} < ticket_J$ then eventually the resource(s) will eventually execute all tasks of J' and of all jobs with

ticket smaller than $ticket_J$. We end up with the previous case, so T is eventually executed, like we wanted to prove. □

Theorem 2 *If a resource executing a task fails, then the task becomes again ready to be executed* (*Partial correctness*).

Proof A resource r executes a task T inside the transaction in lines 7–11 (Algorithm 2). The theorem is only relevant after the task tuple $\mathcal{T}_T$ is removed from the space in line 8. If r fails, the semantics of the transaction for the *inp* operation is clear: $\mathcal{T}_T$ is returned to the tuple space, like we wanted to prove (Sect. 2). A checkpoint tuple may also be inserted in the space in line 20 and left in the space in case of failure, due to the semantics of top-level nested transactions. However, this does not interfere with the fact that $\mathcal{T}_T$ is returned to the space, so task T becomes again ready to be executed, like we wanted to prove. □

8 Evaluation

It is difficult to compare our infrastructure with the traditional ones. Thus, we compared some scheduling algorithms used in traditional grid infrastructures with a simple algorithm that we developed (ReTaClasses) that is used in GridTS. This section briefly presents an overview of the scheduling algorithms used in the simulations and the experimental results.

8.1 Scheduling algorithms

Workqueue (WQ) is a scheduling algorithm that does not use any information about resources for task scheduling [25]. The first task waiting to be scheduled is picked and a free resource is assigned arbitrarily to execute it. This procedure is repeated until all tasks are scheduled.

Workqueue with Replication (WQR) algorithm does the same as WQ [25]. However, when there are no more tasks to be executed and there are still idle resources, the tasks that are still running are replicated in these idle resources, i.e., they are also executed in these resources. When a task replica terminates, all its replicas are stopped. The idea is that when a task is replicated there is a chance that a replica is assigned to a faster node, thus augmenting the probability of a faster completion of the task.

MFTF (Most Fit Task First) [26] uses dynamic information about the resources and task to do the scheduling. MFTF gives more priority to the task that "fits" better to an available resource. The "fitness" value is defined as follows: $fitness(i, j) = \frac{100000}{1+|W_i/S_j - E_i|}$. W_i is the workload of the ith task. S_j is the CPU speed of the jth resource according to the information service. E_i is the expected execution time of the ith task. W_i/S_j is the estimated execution time of task i using the resource j. $W_i/S_j - E_i$ is the difference of the estimated execution time and expected task execution time. A small difference indicates greater suitability between task and node.

We have created a new scheduling algorithm, which we call *ReTaClasses* (*Re*sources and *Ta*sks in *Classes*). The proposed algorithm is simple and basically consists in classifying both tasks and resources in *classes*. Resources are classified in classes $\mathcal{CR} = \{r_1, \ldots, r_{nc}\}$ according to their speed. For instance, if they are classified in three classes ($nc = 3$), the first class can have resources until 1 GHz, the second one resources from 1 to 3 GHz, and the third one over 3 GHz. Tasks are classified in classes $\mathcal{CT} = \{t_1, \ldots, t_{nc}\}$ according to their size. It is the broker that is responsible for classifying the tasks in classes, putting this information in the task tuple (in the *information* field). The number of task and resource classes rc is the same and there is a correspondence between classes: class r_1 should include the slower resources and class t_1 the smaller tasks; class r_{nc} should include the faster resources and class t_{nc} the larger tasks.

Using the ReTaClasses scheduling algorithm, resources start getting tasks from the corresponding task class, i.e., if a resource belongs to class r_i it gets a task from class t_i. When there are no more tasks of class t_i, it tries to get a task of class t_{i+1}; if there no tasks from that class, it tries to get from t_{i+2}, etc.; if there are no more from class t_{nc}, then it starts trying to get tasks from class t_{i-1}, t_{i-2}, etc. If there are no more tasks, it means that all job tasks are being (or have been already) executed. By enforcing faster resources to execute larger tasks first, and slow resources to execute smaller tasks first, the probability of large tasks being scheduled to slow resources is reduced, and the job execution tends to finish faster.

8.2 Simulation environments

To perform the simulations, we developed a simulator, called AGRIS (Another Grid Simulator). This simulator was developed based on GridSim toolkit [27], which consists of a simulation framework that provides key features for simulation of distributed applications in computational grids. The AGRIS extends GridSim in order to support the new grid scheduling infrastructure based on tuple spaces. This extension was made through the implementation of a tuple space and the creation of new classes of brokers and resources, since they behave in GRIDTS is completely different from traditional infrastructure scheduling. In addition, AGRIS extends GridSim, to treat resource failures, as well as to provide checkpoint mechanism.

We simulated 2,490 scenarios and repeated each of them 10 times to compare RETACLASSES with three scheduling

Table 2 Tasks' granularity

Means size of tasks	Number of tasks	Task per resource
1,000	6,000	60
2,500	2,400	24
10,000	600	6
25,000	240	2.4

algorithms: WQ, WQR, and MFTF. ReTaClasses was simulated using one, three, and five classes (denoted, respectively, GRIDTS1, GRIDTS3, and GRIDTS5) and WQR using only two replicas (denoted WQR2X). MFTF used accurate information about resources, something difficult to be obtained in the real world. All simulations used the same value for the grid speed, i.e., for the sum of the resources speeds: 1,000. The *resource speed* represents how fast it can execute a task. A resource with speed 5 can execute a task with size 100 in 20 time units. We also used a fixed value for the job size: 6,000,000 time units. In a ideal world, the makespan of this job would be 6,000 time units, i.e., 100 hours, if the unit was the minute. Thus, by fixing the grid speed and the job size, the variation of makespan is due only to the differences of the scheduling algorithms.

In grid computing, the makespan depends on several parameters, like the number of resources and tasks, the *task granularity* (task size), the *tasks heterogeneity* (the variation of the tasks sizes), and the *resources heterogeneity* (the variation of the resources speeds). We also considered the *fault load*, the number of resources failures, since GRIDTS was designed to be fault-tolerant. The combination of these parameters defines specific execution environments.

The grid *resources speed* have a $U(10 - hm/2, 10 + hm/2)$ distribution, where $U(a, b)$ represents an uniform distribution from a to b and the values used for hm were 0, 2, 4, 8, and 16. This means that the average speed of all resources is 10. When $hm = 0$, all resources have speed 10, so the grid is homogeneous. The maximum heterogeneity of the resources happens when $hm = 16$ and the speed of the resources varies with distribution $U(2, 18)$. Concerning the *tasks granularity*, the experiments considered four groups of task sizes with mean sizes of 1,000, 2,500, 10,000 and 25,000 time units. The higher is the mean size of the tasks, the smaller is the number of tasks per resource, as can be observed in Table 2. It can be observed in Table 2 when the mean task size is 1,000, there are 6,000 tasks and 60 tasks per resource on average, and when the mean task size is 25,000, there are 240 tasks and 2.4 tasks per resource.

To simulate the *heterogeneity of tasks*, in each group, the task sizes were varied 0%, 25%, 50%, 75%, and 100%. A variation of 0% means all tasks have the same size (homogenous job), while a variation of 50% means the tasks sizes have a uniform distribution $U(7,500, 12,500)$. The

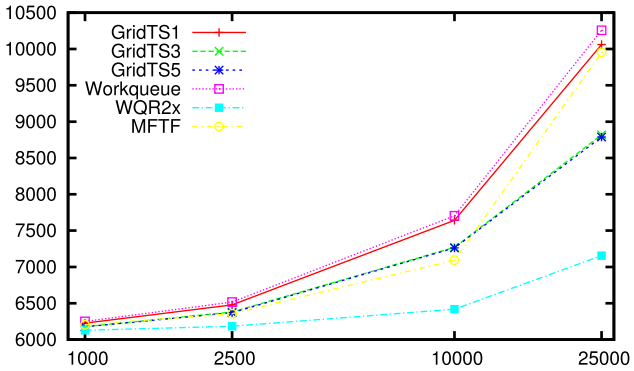

Fig. 3 Average makespan varying the task granularity (no failures)

fault load defines the faults occurred in the system during the execution of a job. In the *failure-free* fault load, there are no failures, i.e., all resources behave correctly. In the *fail-stop* fault load, a percentage of the resources crash during the simulation.

8.3 Simulation without failures

Figure 3 shows the average makespan with different mean task sizes (1,000, 2,500, 10,000, 25,000). Each point was obtained as the average of all levels of tasks and resources heterogeneity. It can be observed when tasks are smaller, the schedulers tend to have similar performance. The reason for this behavior is that there are many tasks per resource, so all resources tend to be busy all the time. However, as the size of tasks grows, differences among schedulers makespan increase.

As we expected, GRIDTS1 had similar performance to WQ. With larger tasks, both had the highest makespan, since large tasks can be scheduled to slow resources near the end of the job, taking more time to terminate. The figure shows that the use of classes in GRIDTS minimizes this effect (GRIDTS3, GRIDTS5). Enforcing resources to execute tasks of the most fit class first, the probability of a larger task being scheduled to slow resources becomes smaller. GRIDTS is better when the number of tasks executed per resource is high. WQR has better performance than the other schedulers because, at the end of simulation, it replicates the tasks to available resources. This approach, however, has no impact when successive jobs are being scheduled. Also, when tasks become large—less tasks per resource—the performance of WQR starts to decrease. The reason is that a large task and its replicas can be scheduled to slow resources, harming the job execution time.

MFTF has good performance only when tasks are small. The justification for this is that MFTF assigns a task to the most suitable resource, but it may not be the fastest resource available. Therefore, the solution chosen by the scheduler may not lead to the best makespan, but it can get stable execution times similar to the expected execution time (E_i) of

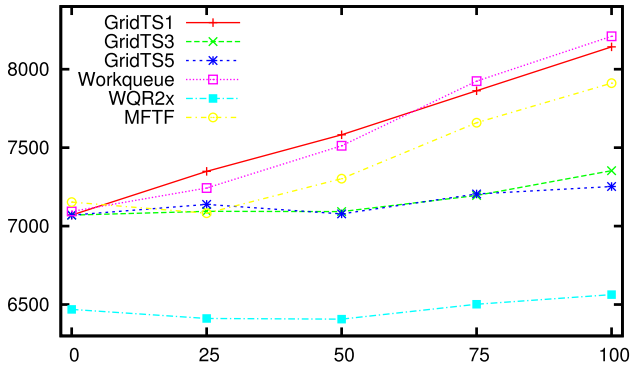

Fig. 4 Average makespan varying the task heterogeneity (no failures)

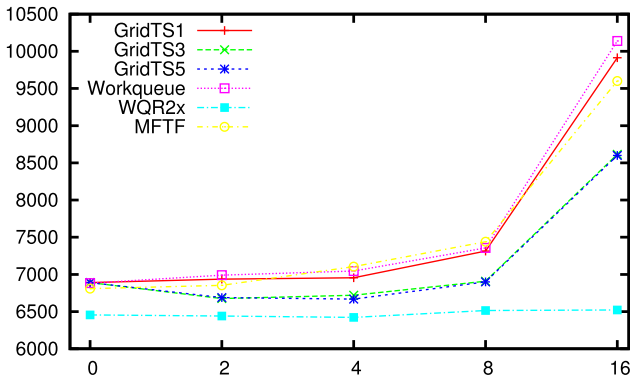

Fig. 5 Average makespan varying the resource heterogeneity (no failures)

each task. The fitness of a task to a resource depends on E_i, so calculating E_i is crucial for getting the best makespan possible, but in practice it is hard to obtain. In the simulations, we set E_i to the mean task size divided by the mean resource speed. When tasks are larger, the task sizes heterogeneity leads to a higher distance between the ratio workload/speed and E_i, leading to lower fitness values. Therefore, tasks that are much larger or much smaller than the mean task size get worse fitness values, which harm the scheduling.

Figure 4 evaluates how each scheduler behaves with different levels of tasks heterogeneity. Like the previous figure, each point is the average of all levels of resources heterogeneity and tasks granularity. Like before, GRIDTS1 has similar performance to WQ. The performance of WQR remains almost unaltered in all cases, due to its replication scheme. Using classes with GRIDTS (GRIDTS3 and GRIDTS5), makes its performance stay almost unaltered like WQR. The performance achieved by GRIDTS3 and GRIDTS5 can be credited to the ability of a powerful resource to choose a large task to execute. The performance of MFTF becomes worse when the tasks heterogeneity augments, for the reasons discussed above: the higher the difference among tasks sizes, the worse the fitness and the higher the job execution time.

Figure 5 evaluates how each scheduler behaves with different levels of resources heterogeneity. As before, each point is the average of all levels of tasks heterogeneity and tasks granularity. Again, GRIDTS1 has similar performance to WQ. WQR performance stays almost unaltered in all cases. The performance of GRIDTS classes GRIDTS3 and GRIDTS5 stays almost unaltered while the resources heterogeneity level is less or equal to 8. With level 16, their performance degrades. MFTF presents the same performance as before.

8.4 Simulation with failures

This section presents the behavior of the algorithms when subject to different fault loads. The experiments were carried out by having a percentage of resources failing, i.e., stopping to execute, at random time during the job execution. For GRIDTS is shown the results using only three classes. In the experiments, each point is the average for all levels of resources heterogeneity. Figure 6 shows three different levels of tasks granularity (2500, 10000, 25000), varying 50% among task sizes in each level.

When there are more than 50% of the resources subject to failures, the performance of GRIDTS3 becomes better than WQR. The reason for this behavior is that when there are many failed resources, the chance of a resource being available to replicate tasks decreases, so WQR starts behaving like Workqueue. Similarly to fault-free environments, MFTF does not have good performance in environments subject to failures. Again, this is due to the difficulty in calculating a good value for E_i. It was calculated without considering faults in the system, since it is not clear how this information might be included in the calculation.

8.5 Summary of the evaluation

The simulations lead us to some interesting conclusions. The first one is that GRIDTS with 3 or 5 classes of tasks/resources has better makespan than most of the other algorithms, with the exception of WQR when the number of resources failures is not high (in this case GRIDTS is also better than WQR). However, in the simulations WQR benefited from the fact that each simulation was for a single job, so WQR had the opportunity of using additional resources to replicate tasks and reduce the makespan. However, in grids permanently executing jobs, it is difficult to happen.

It is especially interesting that GRIDTS had better performance than MFTF because the nontrivial definition of a parameter (E_i), while GRIDTS does not have this difficulty. Another interesting conclusion is that the performance of GRIDTS improves if there are 3 classes instead of just one, but is similar with 3 and 5 classes, so apparently there is no benefit in having more than 3 classes.

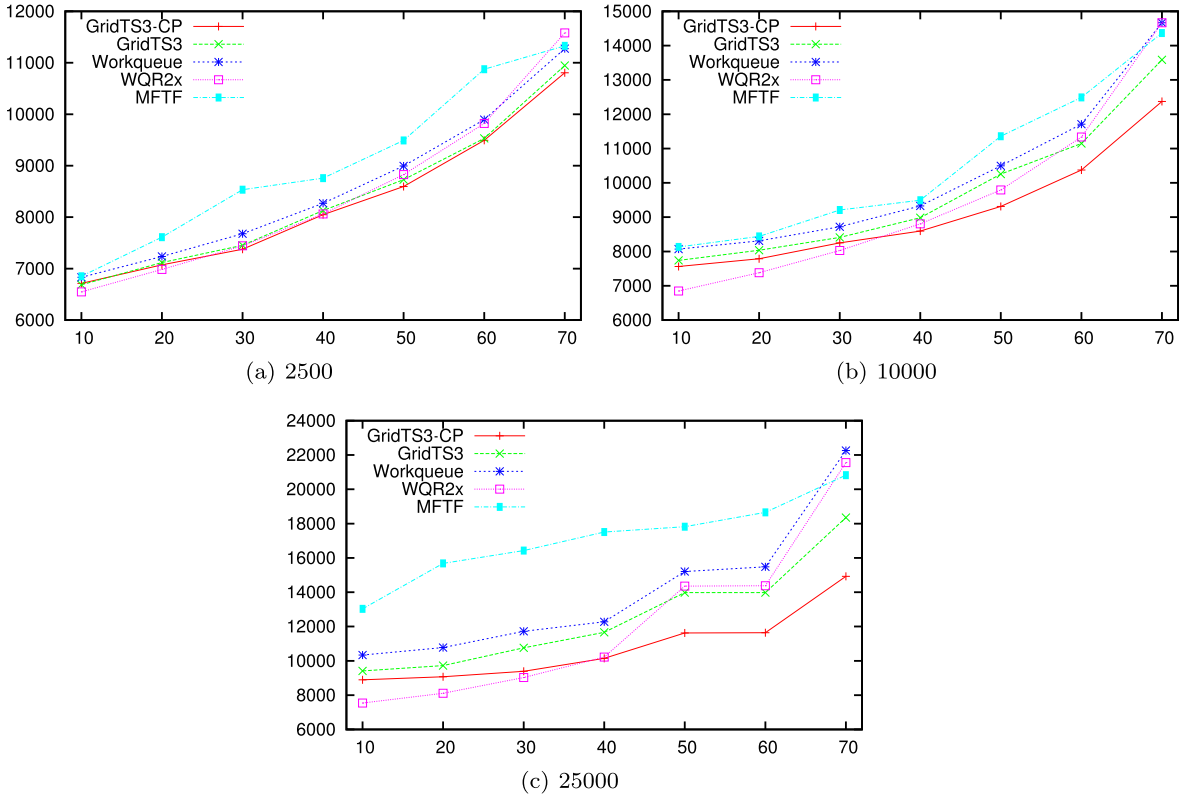

Fig. 6 Average makespan considering failures

Finally, the simulations confirm the expected result that the fault tolerant mechanism has a positive effect in the makespan when there are resource failures, more when the number of failures is higher. The simulations performed do not permit to see the benefits of always having fresh information about the resources (in GRIDTS) over having information that may be somewhat old or hard to collect (in the knowledge-based schedulers—MFTF).

9 Related work

Related work on scheduling algorithms was already presented in Sect. 8.1. Here, we discuss briefly a related work.

TaskSpaces is a framework for grid computing [28]. The paper claims that the framework is based on tuple spaces but it does not seem to be true: tuple spaces partially inspired the approach but TaskSpaces ends up using a communication mechanism similar to message queues. TaskSpaces uses two "tuple space" instances, one for tasks that is called *task bag* and other for results that is called *result bag*. The application sends the tasks to the task bag, and the task bag sends those tasks to the resources. After executing a task, the resource puts a result in the result bag, from which the results are taken by the users. TaskSpaces uses an event notification model where resources register with the task bag in order to receive tasks. The TaskSpaces acts like a "superqueue" where tasks are inserted and then are forwarded to the registered resources in a FIFO order, there is no information about resources or tasks in this process. Thus, TaskSpaces does not make their decisions with any kind of information. If tasks being executed in different resources need to communicate, they exchange information about their IP address and ports through yet another tuple space instance, called communication bag. TaskSpaces is not fault-tolerant.

PLinda [29] and FT-Linda [15] are fault tolerant extensions of the Linda language. PLinda uses a checkpoint mechanism to tolerate faults on the tuple space, and uses a transaction mechanism to allow processes to execute multiple tuple space operations atomically. FT-Linda assumes a set of replicated tuple spaces interconnected by a network supporting total order broadcast [30]. FT-Linda has a restricted form of transactions mechanism called atomic guarded statements (AGS). AGSs can execute multiple tuple space operations atomically, but do not allow computation between the operations. Both Plinda and FT-Linda use the replicated-worker pattern to exemplify the use of their fault tolerance extensions in a cluster environment. In both, Plinda and FT-Linda, the execution of a task by a resource is done inside a transaction context (or AGS context). Thus, if a resource fails while executing a task, the task can be executed by another resource, but all processing executed

is lost, because the state of the process is only saved after the transaction is committed. Our approach also executes a task within a transaction context, but we use a checkpointing mechanism that allows a task to be resumed from the last checkpoint saved in case of failure. Moreover, these works do not deal with the problem of fairness in job execution.

10 Conclusions

This paper presented GRIDTS, a new grid scheduling fault-tolerant infrastructure. The scheduling task model provided by GRIDTS takes from broker the concern to know what resources the tasks will be executed. Moreover, GRIDTS has the immediate benefit of not requiring an information service for indicating the resource utilization and even get good schedules. On the contrary, it leverages naturally the scheduling completely decentralized and it enforces a natural form of load balancing since the resources pick tasks adequate to their conditions and get a new one whenever the previous ended.

To provide this type of scheduling, GRIDTS had to deal with two challenges: fair and fault-tolerant scheduling. Fair scheduling is provided through the FIFO-Except criteria, also proposed in this paper. Fault-tolerant scheduling is provided by combining different fault tolerance techniques: checkpointing, transactions, and replication. The paper presented GRIDTS in detail, including the algorithms executed by the brokers and resources.

This paper also presented several results that show GRIDTS is a highly practicable solution to grid computing. Such results were obtained from correctness proofs and also through the experimental performance evaluation of a scheduling algorithm—ReTaClasses—developed to use the new scheduling infrastructure proposed. Simulation results shows that GRIDTS can be implemented and it solves the problems of obtaining up-to-date information about grid resources.

Acknowledgements This work was supported by Fundação Araucária and by the Program for Research Support of UTFPR—Campus Pato Branco, Directorate of Research and Post-Graduation (DIRPPG). We thank our colleagues Alysson Bessani and Eduardo Alchieri for many discussions on the topics of this article. The authors also thank the anonymous reviewers for their comments.

References

1. Foster I, Kesselman C (1997) Int J Supercomput Appl High Perform Comput 11(2):115
2. Chandy KM, Lamport L (1985) ACM Trans Comput Syst 3(1):63
3. Long D, Muir A, Golding R (1995) In: SRDS'95: Proceedings of the 14TH symposium on reliable distributed systems, IEEE Comput Soc, Washington, p 2
4. Gelernter D (1985) ACM Trans Program Lang Syst 7(1):80
5. Cabri G, Leonardi L, Zambonelli F (2000) IEEE Comput 33(2):82
6. Rowstron AIT, Wood A (1998) Sci Comput Program 31(2–3):335
7. Schopf JM (2004) In: Nabrzyski J, Schopf JM, Weglarz J (eds) Grid resource management: state of the art and future trends. Kluwer Academic, Norwell, pp 15–23
8. Carriero N, Gelernter D (1989) Commun ACM 32(4):444
9. Smith JA, Shrivastava SK (1996) In: 2nd international Euro-Par conference, pp 487–495
10. Bessani AN, Alchieri E, Correia M, Fraga JS (2008) In: Proceedings of the 3rd ACM/SIGOPS/EuroSys European conference on computer systems (EuroSys), p 2008
11. Castro M, Liskov B (2002) ACM Trans Comput Syst 20(4):398
12. Dwork C, Lynch NA, Stockmeyer L (1988) J ACM 35(2):288
13. Haerder T, Reuter A (1983) ACM Comput Surv 15(4):287
14. Xu A, Liskov B (1989) In: 19th symposium on fault-tolerant computing (FTCS'89), pp 199–206
15. Bakken DE, Schlichting RD (1995) IEEE Trans Parallel Distrib Syst 06(3):287. http://doi.ieeecomputersociety.org/10.1109/71.372777
16. Obelheiro RR, Bessani AN, Lung LC, Correia M (2006) How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep of Informatics, University of Lisbon
17. Andrade N, Cirne W, Brasileiro F, Roisenberg P (2003) In: Job scheduling strategies for parallel processing, pp 61–86. Lecture notes computer science, vol 2862. Springer, Berlin
18. Lehman TJ, Cozzi A, Xiong Y, Gottschalk J, Vasudevan V, Landis S, Davis P, Khavar B, Bowman P (2001) Comput Netw 35(4):457. doi:10.1016/S1389-1286(00)00178-X
19. Koo R, Toueg S (1987) IEEE Trans Softw Eng 13(1):23
20. Breg F, Polychronopoulos C (2001) In: Proceedings of the ACM 2001 Java Grande/ISCOPE conference, Palo Alto, Calif, June 2–4, 2001. ACM, New York, pp 173–180
21. Microsystems Sun Javaspaces service specification. Available in http://www.jini.org/wiki/JavaSpaces_Specification 2003
22. Favarim F, Alchieri E, da Silva Fraga J, Lung L, Bessani AN (2009) In: 8th international workshop on the foundations of coordination languages and software architectures (FOCLASA)
23. Gray J (1978) In: Operating systems, an advanced course, pp 393–481
24. Kung HT, Robinson JT (1981) ACM Trans Database Syst 6(2):213. doi:10.1145/319566.319567
25. Silva D, Cirne W, Brasileiro FV (2003) In: 9th international Euro-Par conference, pp 169–180
26. Wang SD, Hsu IT, Huang ZY (2005) In: 11th international conference on parallel and distributed systems, pp 22–28. doi:10.1109/ICPADS.2005.138
27. Buyya R, Murshed M (2002) J Concurr Comput, Pract Exp (CCPE) 14(13–15):1175
28. Sterck HD, Markel RS, Phol T, Rde U (2003) In: ACM symposium on applied computing, pp 1024–1030
29. Jeong K, Shasha D (1994) In: Proceedings of the 13th symposium on reliable distributed systems, pp 96–105
30. Hadzilacos V, Toueg S (1994) A modular approach to the specification and implementation of fault-tolerant broadcasts. Tech Rep TR 94-1425, Department of Computer Science, Cornell University